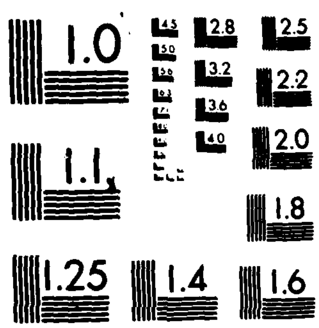
AD-A139 307 SOLVING THE POISSON EQUATION ON THE FPS-164 (FLOATING POINT SYSTEM-164)(U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE S T O'DONNELL ET AL. NOV 83 1/1

UNCLASSIFIED YALEU/DCS/TR-293 N00014-82-K-0184 F/G 12/1 NL

END
DATE
FILMED
4-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LUX ET VERITAS

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

SOLVING THE POISSON EQUATION ON THE FPS-164

Susan Temple O'Donnell[1]
Peter Geiger[2]
Martin H. Schultz[1]

November 1983

Technical Report #293

[1]Research Center for Scientific Computation, Department of Computer Science, Yale University, Box 2158, Yale Station, New Haven, Conn 06520.

[2]Swiss Federal Institut of Technologie, Department of Applied Mathematics, 8092 Zuerich, Switzerland.

This work was supported in part by ONR Grant #N00014-82-K-0184, NSF Grant #MCS-8106181 and External Research Grants from Digital Equipment Corporation and Floating Point Systems.

1. Introduction

The architectural differences between a serial and a parallel machine raise a number of questions regarding the efficiency of established algorithms. In this paper we explore several algorithms which solve the Poisson equation on rectangular regions in two dimensions. The solution of the Poisson problem is an example of one of the simplest nontrivial computations which frequently occur in innermost loops of large scale scientific codes, and hence is a useful test of different architectures for scientific computation. Compared are ~~We compare~~ solution times on the Vax 11/780 with solution times on the Floating Point System 164 (FPS-164) attached processor.[*] Since the FPS-164 supports a sufficiently large memory and the host/attached processor I/O is relatively slow, it is of interest to solve large problems entirely on the FPS-164. We explore the performance of the FPS-164 on both portable FORTRAN programs which have not been tuned to its architecture and on moderately tuned FORTRAN programs which make calls to the FPS assembly language math library, MATHLIB. Use of MATHLIB results in shorter programs which are usually more efficient. We show that the speedup in execution time is more uniform across the algorithms than might be anticipated and hence the choice of algorithm is still highly significant.

In Section 2 we outline three standard algorithms, the complete Fourier algorithm based on double Fourier transforms, the Fourier/Tridiagonal algorithm and the FACR(1) algorithm. In Section 3 we examine the tridiagonal linear system solution, which is part of the latter two algorithms, in more detail. We analyze the classical Gauss algorithm and examine a refinement of the tridiagonal linear system solution based on ideas of Malcolm and Palmer [9], which appear to be too little known [12]. In Section 4, we consider optimizations of Cooley's [2] method for calculating the sine transform as a real transform of half size. We present timings from our implementation of each of the algorithms for the Poisson problem in Section 5. Finally, we give concluding remarks in Section 6.

[*]The Floating Point System 164 attached processor (AP) is a 64 bit, 11-MFLOP machine with a 182-nanosecond cycle time. Multiple functional units together allow as many as 10 simultaneous operations.

2. Three Basic Algorithms

We begin by describing some fast direct methods which are commonly used to solve the Poisson equation

$$-\Delta u = f \tag{1}$$

with Dirichlet boundary conditions in two dimensional rectangles.

For simplicity in notation we consider only the case of the square. Define the grid points $(x_i,y_k) = (ih,kh)$ for $i,k = 0,\ldots,n+1$ with $h = 1/(n+1)$ and the corresponding function values $u_{ik} \equiv u(x_i,y_k)$ and $f_{ik} \equiv f(x_i,y_k)$. We consider the simple 5-point discrete Poisson equation

$$-\Delta^{(h)} u_{ik} = f_{ik}, \qquad i,k = 1,\ldots,n, \tag{2}$$

where the 5-point Laplacian operator $\Delta^{(h)}$ is defined by

$$\Delta^{(h)} u_{ik} \equiv \frac{1}{h^2} [-4u_{ik} + u_{i-1,k} + u_{i+1,k} + u_{i,k-1} + u_{i,k+1}] \tag{3}$$

subject to the boundary conditions

$$u_{0k} = u_{n+1,k} = u_{i0} = u_{i,n+1} = 0, \qquad i,k = 1,\ldots,n. \tag{4}$$

In Sections 2.1-2.3, we describe three well known fast direct methods for solving the discrete Poisson equation.

2.1. The Complete Fourier algorithm

The approach of using Fast Fourier techniques to solve the above finite difference equations originates with Hockney [8]. The first method we describe involves Fourier transforms in each variable.

We extend the sequences $u \equiv \{u_{ik}\}$ and $f \equiv \{f_{ik}\}$ to be odd doubly periodic sequences of period $2(n+1)$ in both variables. This is valid since u satisfies the boundary values (4) and we

may set $f_{0k} = f_{n+1,k} = f_{i0} = f_{i,n+1} = 0$ since equation (2) does not restrict the values of f on the boundary.

For the 5-point Laplacian operator, let $d \equiv \{d_{ik}\}$ be the doubly periodic sequence of period $2(n+1)$ defined by

$$d_{00} = 4, \qquad d_{-1,0} = d_{1,0} = d_{0,-1} = d_{0,1} = -1, \qquad d_{ik} = 0 \ \text{otherwise.}$$

With this notation, we may rewrite equation (2) as the convolution

$$d * u = h^2 f \tag{5}$$

which can be solved for u by using discrete double Fourier transforms.

Let $\hat{x}$ denote the discrete Fourier transform of the doubly periodic sequence x with period $2(n+1)$ in both variables. Equation (5) then implies

$$\hat{d} \cdot \hat{u} = h^2 \hat{f} \,. \tag{6}$$

The Fourier transform $\hat{d}$ of d can easily be calculated to be

$$\hat{d}_{ik} = \frac{1}{4(n+1)^2} \left[1 - \frac{1}{2} \cos\left(\frac{i\pi}{n+1}\right) - \frac{1}{2} \cos\left(\frac{k\pi}{n+1}\right) \right]$$

and therefore we can solve for $\hat{u}_{ik}$:

$$\hat{u}_{ik} = h^2 \ \hat{d}_{ik}^{-1} \ \hat{f}_{ik} \,. \tag{7}$$

The discretization of the Poisson problem, represented by equation (2) can now be solved by the following algorithm:

1. Calculate the Fourier transform $\hat{f}$.
2. Calculate $\hat{u}$ by (7).
3. Perform the inverse Fourier transform to recover u from $\hat{u}$.

We remark that since $\hat{u}$ is real and odd, it suffices in steps 1 and 2 to use a sine transform in place of a Fourier transform.

The complete Fourier algorithm also provides an easy way to solve the 9-point discretization and other more precise approximations to equation (2), see Henrici [7]. For example, if we use the fourth order 9-point discretization for the Laplacian

$$\Delta_9^{(h)} u_{ik} \equiv \frac{1}{h^2} [-20u_{ik} + 4u_{i-1,k} + 4u_{i+1,k} + 4u_{i,k-1} + 4u_{i,k+1} + u_{i-1,k-1} + u_{i-1,k+1} + u_{i+1,k-1} + u_{i+1,k+1}] \tag{8}$$

and for the right hand side of equation (2) we use the mean values $g \equiv \{g_{ik}\}$, where

$$g_{ik} \equiv \frac{1}{12} (8f_{ik} + f_{i-1,k} + f_{i+1,k} + f_{i,k-1} + f_{i,k+1}) \, , \tag{9}$$

we get an equation of the same form as (5) to solve:

$$d * u = h^2 g \, . \tag{10}$$

In this case, we have

$$d_{00} = 20, \qquad d_{-1,0} = d_{1,0} = d_{0,-1} = d_{0,1} = -4,$$
$$d_{-1,-1} = d_{-1,1} = d_{1,-1} = d_{1,1} = -1, \text{ and } d_{ik} = 0 \text{ otherwise.}$$

2.2. The Fourier/Tridiagonal algorithm

In the literature, this algorithm is often simply called the Fourier or basic Fourier algorithm, see Temperton [12]. It is based on matrix decomposition, see Buzbee, Golub and Nielson [1].

The discretized Poisson equation (2)-(3), can written in matrix form:

$$M u = h^2 f \equiv y \tag{11}$$

where the vector u (similarly f and y) may be written

$$u \equiv \begin{bmatrix} u_1 \\ u_2 \\ \cdot \\ \cdot \\ u_n \end{bmatrix} \quad \text{and} \quad u_i \equiv \begin{bmatrix} u_{i1} \\ u_{i2} \\ \cdot \\ \cdot \\ u_{in} \end{bmatrix}$$

and the $n^2 \times n^2$ matrix M is block tridiagonal of block order n

$$M \equiv \begin{bmatrix} A & I & & & \\ I & A & I & & O \\ & \cdot & \cdot & \cdot & \\ O & & I & A & I \\ & & & I & A \end{bmatrix} , \tag{12}$$

where A is the tridiagonal n×n matrix is defined by

$$A \equiv \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & O \\ & \cdot & \cdot & \cdot & \\ O & & 1 & -4 & 1 \\ & & & 1 & -4 \end{bmatrix} . \tag{13}$$

The eigenvalues λ_k and eigenvectors $v_k \equiv \{v_{ik}\}$ of A are known to be

$$\lambda_k = -4 + 2\cos\left(\frac{k\pi}{n+1}\right), \qquad k = 1,\ldots,n , \tag{14}$$

$$v_{ik} = \sqrt{\frac{1}{n+1}}\ \sin\left(\frac{ik\pi}{n+1}\right), \qquad i,k = 1,\ldots,n . \tag{15}$$

With $V^T \equiv V \equiv \{v_{ik}\}$ and Λ the diagonal matrix diag(λ_k), we have

$$VAV = \Lambda . \tag{16}$$

We may rewrite equation (11) as

$$\begin{aligned} Au_1 + u_2 &= y_1 , \\ u_{i-1} + Au_i + u_{i+1} &= y_i , \qquad i = 2,\ldots,n-1 \\ u_{n-1} + Au_n &= y_n . \end{aligned} \tag{17}$$

and with (16) this becomes

$$\begin{aligned} \Lambda\tilde{u}_1 + \tilde{u}_2 &= \tilde{y}_1 , \\ \tilde{u}_{i-1} + \Lambda\tilde{u}_i + \tilde{u}_{i+1} &= \tilde{y}_i , \qquad i = 2,\ldots,n-1 \\ \tilde{u}_{n-1} + \Lambda\tilde{u}_n &= \tilde{y}_n . \end{aligned} \tag{18}$$

where

$$\tilde{u}_i = Vu_i , \tag{19}$$

and

$$\tilde{y}_i = Vy_i . \tag{20}$$

If we rearrange the equations (18), we get for each $k = 1,\ldots,n$

$$\begin{aligned} \lambda_k\tilde{u}_{1k} + \tilde{u}_{2k} &= \tilde{y}_{1k} , \\ \tilde{u}_{i-1,k} + \lambda_k\tilde{u}_{ik} + \tilde{u}_{i+1,k} &= \tilde{y}_{ik} , \qquad i = 2,\ldots,n-1 \\ \tilde{u}_{n-1,k} + \lambda_k\tilde{u}_{nk} &= \tilde{y}_{nk} . \end{aligned} \tag{21}$$

Thus we get the following algorithm:

1. Perform the transformation (20).
2. Solve the n tridiagonal linear systems (21).
3. Solve (19) for u_i, i.e. perform the inverse transform.

2.3. The FACR(1) algorithm

Following Hockney [8], we start by eliminating the vectors u_i with odd indices from equation (17), i.e., considering u as a matrix, we eliminate the odd rows. We could also eliminate the odd columns of u, f, see Temperton [12].

In fact, for indices $i = 4,6,\ldots,n-3$,

$$
\begin{aligned}
u_{i-2} + Au_{i-1} + u_i \quad & = y_{i-1} \ , \\
u_{i-1} + Au_i + u_{i+1} \quad & = y_i \ , \\
u_i + Au_{i+1} + u_{i+2} & = y_{i+1} \ ,
\end{aligned}
\tag{22}
$$

and similar equations for $i = 2$ and $i = n-1$. Multiplying the middle equation of 22 by -A and adding all three equations, we get

$$u_{i-2} + (2I-A^2)u_i + u_{i-2} = y_{i-1} - Ay_i + y_{i+1}$$

and hence the following system, where we assume that n is odd. (For the case where n is even, the last equation will differ.)

$$
\begin{bmatrix}
2I-A^2 & I & & & \\
I & 2I-A^2 & I & & \\
 & \cdot & \cdot & \cdot & \\
 & & I & 2I-A^2 & I \\
 & & & I & 2I-A^2
\end{bmatrix}
\begin{bmatrix} u_2 \\ u_4 \\ \cdot \\ u_{n-3} \\ u_{n-1} \end{bmatrix}
=
\begin{bmatrix}
y_1 + y_3 - Ay_2 \\
y_3 + y_5 - Ay_4 \\
\cdot \\
y_{n-4} + y_{n-2} - Ay_{n-3} \\
y_{n-2} + y_n - Ay_{n-1}
\end{bmatrix}
\tag{23}
$$

Continuing this step of eliminating of every other row leads to the algorithm of cyclic reduction. In the literature this is sometimes refered to as the FACR (Fourier and Cyclic Reduction) algorithm. We stop after one reduction step and use the Fourier/Tridiagonal algorithm to solve system (23). *This is called the* FACR(1) algorithm. We replace matrix A of Section 2.2 with the matrix $2I - A^2$. Since $B = 2I - A^2$ is a polynomial $p = p(A)$ in A, it has the same eigenvectors as A and eigenvalues $\lambda_B = p(\lambda_A) = 2 - \lambda_A^{\ 2}$. The solution on the odd rows is obtained by solving the tridiagonal system corresponding to the middle equation of (22) for u_i:

$$Au_1 = y_1 - u_2 \,,$$

$$Au_i = y_i - u_{i-1} - u_{i+1} \,, \qquad i = 3,5,\ldots,n-2, \tag{24}$$

$$Au_n = y_n - u_{n-1} \,.$$

The algorithm is summarized as follows:

1. Set up the right hand side of (23).
2. Solve (23) by the algorithm from (2.2).
3. Calculate the solution on the odd rows from (24).

3. Refinement of the Tridiagonal Linear System Solution

Two of the algorithms for solving the Poisson equation in rectangular regions require solving many tridiagonal systems of linear equations.

The problem to be solved is of the form

$$A\,x = y \tag{25}$$

where A is an n×n tridiagonal matrix of the form

$$A = \begin{bmatrix} \lambda & 1 & & & \\ 1 & \lambda & 1 & O & \\ & \cdot & \cdot & \cdot & \\ O & 1 & \lambda & 1 & \\ & & & 1 & \lambda \end{bmatrix} \tag{26}$$

For all λ we have $|\lambda| > 2$. Therefore the system is definite and we are assured of the existence of a solution.

Basically there are two important efficiency issues: (1) avoiding divisions, and (2) keeping the pipelines filled. In Section 3.1 we discuss Gauss elimination and in Section 3.2 we describe the improvements of the Malcolm and Palmer method [9].

3.1. The classic Gauss algorithm

The easiest way so solve the system (25) is the well known Gauss elimination in three steps:

1. LR-factorization: $A = LR$[4] with

$$L = \begin{bmatrix} l_1 & & & & \\ 1 & l_2 & & O & \\ & \cdot & \cdot & & \\ O & & 1 & l_{n-1} & \\ & & & 1 & l_n \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 1 & r_1 & & & \\ & 1 & r_2 & & O \\ & & \cdot & \cdot & \\ & O & & 1 & r_{n-1} \\ & & & & 1 \end{bmatrix}. \tag{27}$$

[4]We use notation $A = LR$ in preference to the usual LU notation to avoid confusion with the u from Section 2.

From $A = LR$ we get recursion formulas for the elements l_k and r_k:

$$l_1 = \lambda, \qquad \left.\begin{array}{l} l_k r_k = 1 \\ l_{k+1} + r_k = \lambda \end{array}\right\}, \qquad k = 1, \ldots, n.$$

2. Forward substitution: $Lw = y$,

$$w_1 = \frac{y_1}{l_1}, \qquad w_k = \frac{y_k - w_{k-1}}{l_k}, \qquad k = 2, \ldots, n.$$

3. Backward substitution: $Rx = w$

$$x_n = w_n \qquad x_k = w_k - r_k x_{k+1} \qquad k = n-1, \ldots, 1$$

This algorithm requires n multiplications, 2n divisions and 3n additions/subtractions.

Since l_k and r_k are reciprocal values we need only one of them. To avoid as many divisions as possible we eliminate l_k. If we define $r_n \equiv 1/l_n$ we get:

$$1. \quad r_1 = \frac{1}{\lambda}, \qquad r_k = \frac{1}{\lambda - r_{k-1}}, \qquad k = 2, \ldots, n. \tag{28}$$

$$2. \quad w_1 = r_1 y_1, \qquad w_k = r_k (y_k - w_{k-1}), \qquad k = 2, \ldots, n. \tag{29}$$

$$3. \quad x_n = w_n, \qquad x_k = w_k - r_k x_{k+1}, \qquad k = n-1, \ldots, 1. \tag{30}$$

This form of the algorithm needs 2n multiplications, n divisions, and 3n additions/subtractions.

In applications where we have to solve several Poisson equations with the same mesh size, we could precompute the r_k, but this would double the amount of storage.

3.2. A more efficient algorithm to solve the tridiagonal systems

There exist several algorithms which improve the storage requirements of the tridiagonal solver. The algorithm of Evans [3] and Evans and Hatzopoulos [4] reduces the number of stored elements r_k from n to n/2. Fischer et al [5] use a Fourier-Toeplitz method which does not need to store any of the r_k but which requires more operations.

On many machines, reducing the division count improves the efficiency of an algorithm. This is particularly true on the FPS-164, as division is done in software and consumes much more time than any other basic operation. The relevant times are

APFTN single division	29 cycles
APFTN vector division	18 cycles/element
APAL single division	22 cycles
APAL vector division	7 cycles/element

as compared to

multiplication	3 cycles
addition	2 cycles

Note that multiplication and addition operations may be initiated every cycle. Significant savings can therefore be made by reducing the number of divisions.

Malcolm and Palmer [9] derive an algorithm which applies to linear systems with real, symmetric, diagonally dominant, tridiagonal coefficient matrices with constant diagonals. This algorithm needs both fewer operations and much less storage. Malcolm and Palmer demonstrate that the entries l_k and the r_k of Section 3.1 converge, and give lower and upper bounds on the rate of convergence, as a function of λ, to a relative error of ϵ. We use an equivalent upper bound,

$$k_0 \equiv \left\lceil \frac{\log \epsilon}{\log \omega} \right\rceil . \tag{31}$$

where ω is defined by $\omega \equiv z_2/z_1$, and where z_1 and z_2 are the solutions of $z = 1/(\lambda - z)$, the equation satisfied by the limit of r_k, such that $|z_1| > |z_2|$. We derive k_0 in Appendix A. To solve each tridiagonal system, we compute k_0 from equation (31), perform only k_0 steps in the Gaussian elimination and use u_{k_0} for the remaining entries in the factor. It is this method that we use for the tridiagonal solver in our implementation of the Fourier/Tridiagonal and

FACR(1) methods for solving the Poisson problem.

If $|\lambda|$ is not too close to 2 we have remarkably fast convergence, and k_0 is the exact index of convergence. In Table 3-1, we show the actual index of convergence for $\epsilon = 10^{-16}$ for varying values of λ. In Table 3-1, k_{conv} denotes the index where convergence actually takes place, (see Appendix A), and k_0 denotes the estimate from (31).

$\lvert \lambda \rvert$	k_{conv}	k_0
2.0001	1445	1842
2.0010	492	582
2.0100	167	184
2.1000	56	58
2.5000	26	26
3.0000	19	19
4.0000	13	13
5.0000	11	11
6.0000	10	10

Table 3-1: Index of convergence (with $\epsilon = 10^{-16}$) in tridiagonal solver

This gives a dramatic reduction in the number of divisions when we solve Poisson's equation using a n×n grid. In this case we solve n systems of form (25) with

$$\lambda = \lambda_k = -4 + 2\cos\left(\frac{k\pi}{n+1}\right), \qquad k = 1, \ldots, n.$$

In Table 3-2, we display a count of the number of divisions required for problems of varying size.

Due to the cost of evaluating (31), we start to cut the amount of work only when $n \geq 31$, but for larger n we have eliminated almost the entire LR-factorization. Our programs use the improved tridiagonal solver only for problems of size ≥ 31.

The effect of this method on the Fourier/Tridiagonal algorithm for the Poisson problem may be seen by comparing the times spent solving n linear systems in the solution of the Poisson problem. Refer to the Fourier/Tridiagonal algorithm in Section 2.2.

This method largely eliminates any advantage of preprocessing in that few factors need be

n	# divisions in the original algorithm = n^2	# divisions using our index of convergence	%
7	49	49	100.00
15	225	202	89.78
31	961	560	58.27
63	3969	1402	35.32
127	16129	3339	20.70
255	65025	7733	11.89
511	261121	17565	6.73
1023	1046529	39296	3.75
2047	4190209	86919	2.07

Table 3-2: Division count in the two tridiagonal solvers

n	time for the original algorithm	time for new algorithm	%
31	0.0088	0.0091	103
63	0.0388	0.0244	72
127	0.1331	0.0704	53
255	0.5271	0.2196	42
511	2.1036	0.7414	35

Table 3-3: Execution time (sec) to solve n tridiagonal systems in APFTN

computed; hence the need to store precomputed factors is eliminated. In addition, the solution of the tridiagonal systems is no more expensive than the normalization in the complete Fourier method, implying that the Fourier/Tridiagonal method is necessarily faster than the complete Fourier method, regardless of the speed of the FFT.

A second major efficiency issue is pipelining. At cost in storage, we explore the tradeoff of the faster run time that becomes possible by solving 4, 8, 16, 32, or n systems in parallel, performing forward solves for multiple systems followed by the back solves. The results are presented in Section 5.

4. The Fourier Transform

In this section, we compare the efficiency of different Fourier Transforms implemented on the VAX and on the AP in FORTRAN, and on the AP using the FPS APMATH library routines. We used compiler optimization level 3 of the Revision D04 APFTN compiler. We took advantage of the problem being both real and odd to implement the sine transform for the FFT, and we also present results using the real FFT. The APMATH library includes a routine for computing a real FFT, but not a sine transformation. We implement our sine transformation using the algorithm by Cooley [2].

prob size n	VAX (FORTRAN)	AP (FORTRAN)	AP (LIBRARY ROUTINES)
128	0.039	0.0029	0.0014
256	0.08	0.0073	0.0032
512	0.19	0.0150	0.0066
1024	0.39	0.0312	0.0147
2048	0.87	0.0649	0.0292
4096	2.23	0.1519	0.0645
8192	4.68	0.3186	0.1335
16384	10.3	0.6561	0.2913

Table 4-1: Execution time (sec) for one complex FFT

prob size n	VAX (FORTRAN)	AP (FORTRAN)	AP (LIBRARY ROUTINES)
128	0.031	0.0023	0.0009
256	0.04	0.0043	0.0017
512	0.11	0.0098	0.0037
1024	0.24	0.0201	0.0076
2048	0.51	0.0412	0.0166
4096	1.12	0.0836	0.0330
8192	2.72	0.1924	0.0721
16384	5.79	0.3967	0.1487

Table 4-2: Execution time (sec) for one real FFT

...s 4-1, 4-2, and 4-3, we present the times for one FFT for problems of one dimension. Note that the real and the complex FFT require arrays of length double the problem size. The

prob size n	VAX (FORTRAN)	AP (FORTRAN)	AP (Cooley)
128	0.02	0.0015	0.0006
256	0.03	0.0028	0.0013
512	0.05	0.0054	0.0021
1024	0.13	0.0121	0.0046
2048	0.25	0.0244	0.0091
4096	0.56	0.0497	0.0201
8192	1.22	0.1022	0.0400
16384	2.95	0.2275	0.0882

Table 4-3: Execution time (sec) for one sine transform

algorithm for the sine transform presented by Cooley [2] involves pack and unpack operations in addition to a real transform. As the times for our sine transform are significantly higher than the expected time of an APAL version, in our Poisson problem we vectorize the computation of the n sine transforms by packing and unpacking some number of vectors, m, in one loop. This allows the APFTN compiler to take significant advantage of pipelining. The storage cost of performing m sine transforms in parallel is $n \times m/2$. We save .0002 seconds per transform in the 255 by 255 size problem, which reduces the number of seconds per sine transform from .0013 to .001. This compares very respectably to the time for a real transform of .0017 seconds. In the 511 by 511 size problem, the number of seconds per sine transform is reduced from .0021 to .0018. The time for the real transform is .0037 seconds. Our tables in Section 5 refer to the vectorized sine transform by the notation, Vecsin. Unless the vectorized sine method is specifically mentioned, the sine transform we implemented is the non-vectorized implementation of Cooley's algorithm.

Note that as is shown in Table 4-4, the complete Fourier algorithm has the most to gain from an efficient sine transform. In section 5 we show that even with the vectorized sine transform, the complete fourier algorithm is not competitive with the FACR(1) method.

	Fourier	Fourier/Trid	FACR(1)
Sine transf. of length n	4n	2n	n
Trid. systems of order n	0	n	n/2*
Trid. systems of order n/2*	0	0	n

*) n/2 may be either n/2 or n/2 ± 1/2

Table 4-4: Count of components required in each algorithm

5. Numerical Results for the Poisson Problem

We present execution times for the two dimensional Poisson problem using the algorithms described in the previous sections. Each algorithm was applied to discretizations varying in size from 31 x 31 points to 511 x 511 points. Each algorithm was implemented in FORTRAN77 on the VAX 11/780 with a FPA, in APFTN64 on the FPS-164, and in APFTN64 with extensive calls to APMATH library routines from the FPS math library. We used compiler optimization level 3 of the Revision D04 APFTN compiler. The FPS-164 supports *sufficiently* large *memory* that the problems run on the FPS-164 were solved entirely on the FPS-164.

One set of tests was run using the real Fast Fourier Transform for the FFT. A second set of tests was run substituting the sine transformation for the real FFT, cf. Section 4.

The times apply to the solution of the Poisson equation and *do not include the overhead costs* of the transfer of program and data to the AP or of paging on the VAX. These are considered separately. The VAX was run single-user with an unrealistically large working-set to eliminate dependence on working set size.

5.1. Computation times

Our results indicate that the gain from the FPS-164 architecture is more uniform across algorithms than might be anticipated. Graphs for each algorithm are presented in Figure 5-1. The ratios of timings on the two *machines for* the algorithms using the real FFT are presented in Table 5-2.

The results presented in Figure 5-1 and Table 5-1 are for FORTRAN programs which, other than for calls to the MATHLIB, have not been specially tuned for the APFTN compiler. In this table, the sine transform refers to the non-vectorized implementation of Cooley's algorithm. In the remainder of this section, we examine several possible improvements to these three algorithms.

We begin by presenting a breakdown of the percentage of time spent in each section of the programs. (See Table 5-3.)

To determine whether or not the complete Fourier algorithm is necessarily less efficient on the parallel architecture of the AP, we first vectorized the normalization, which reduced the

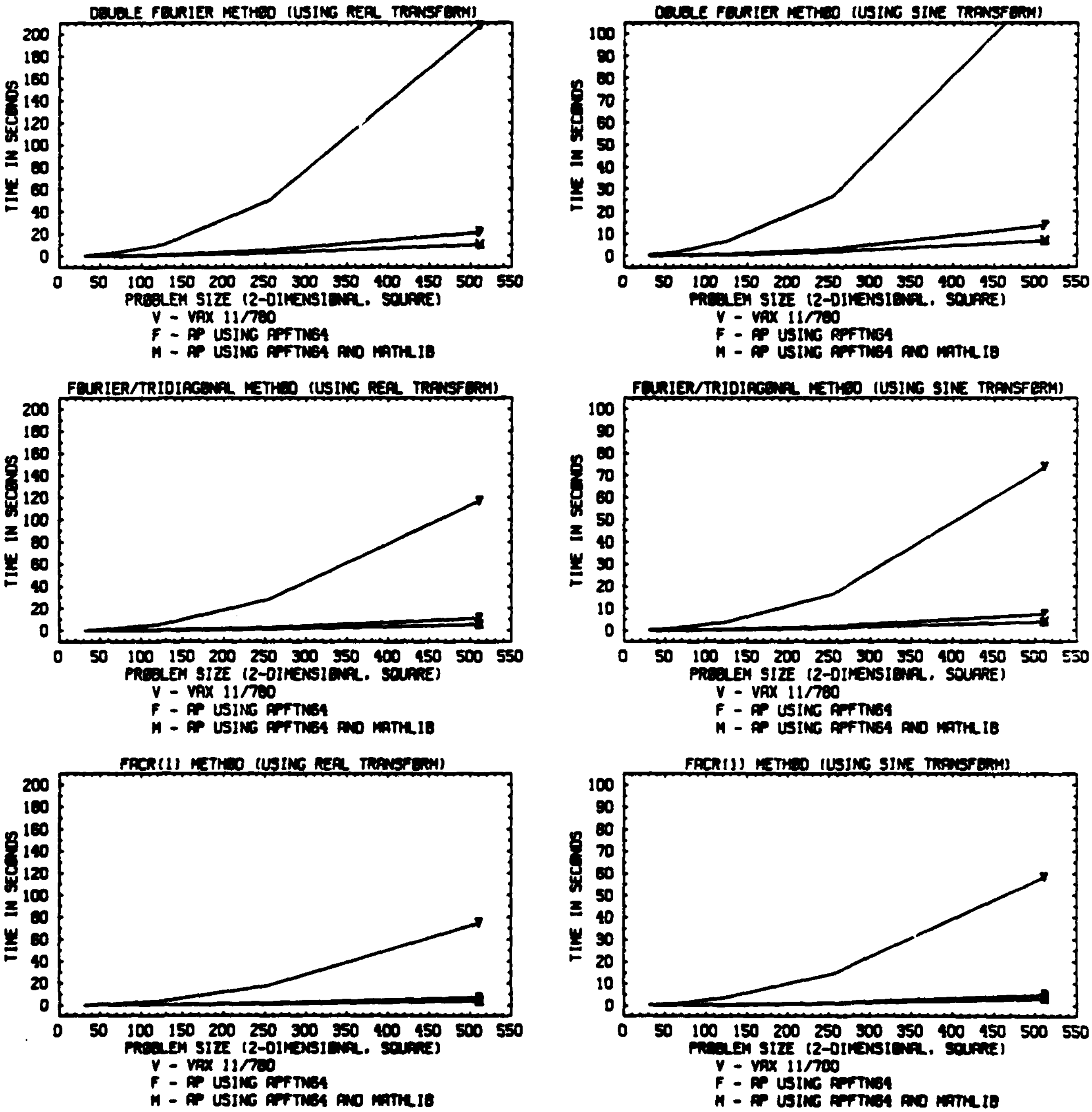


Figure 5-1: Execution times (sec) to solve Poisson's equation

		Poisson Problem					
		Using Real Transform			Using Sine Transform		
		FFT	FFT/Trid	FACR(1)	FFT	FFT/Trid	FACR(1)
	VAX (FORTRAN)	0.62	0.36	0.25	0.44	0.28	0.29
31x31	AP (FORTRAN)	0.10	0.05	0.03	0.08	0.04	0.03
	AP (MATHLIB)	0.04	0.03	0.02	0.04	0.03	0.02
	VAX (FORTRAN)	2.61	1.49	0.99	1.64	1.00	0.73
63x63	AP (FORTRAN)	0.34	0.19	0.12	0.24	0.14	0.10
	AP (MATHLIB)	0.16	0.10	0.08	0.13	0.10	0.08
	VAX (FORTRAN)	10.39	5.99	3.93	6.72	4.12	3.90
127x127	AP (FORTRAN)	1.21	0.67	0.43	0.87	0.50	0.35
	AP (MATHLIB)	0.62	0.37	0.27	0.46	0.29	0.24
	VAX (FORTRAN)	50.33	27.95	17.78	26.86	16.47	14.5
255x255	AP (FORTRAN)	5.30	2.85	1.76	3.20	1.79	1.23
	AP (MATHLIB)	2.42	1.39	1.00	1.77	1.07	0.85
	VAX (FORTRAN)	206.9	116.74	74.84	123.07	73.59	58.16
511x511	AP (FORTRAN)	21.31	11.26	6.89	13.64	7.43	4.99
	AP (MATHLIB)	10.22	5.65	3.97	6.76	3.93	3.10

Table 5-1: Execution times (sec) to solve Poisson's equation

Algorithm	VAX/AP(FTN)	AP(FTN)/AP(MATHLIB)	VAX/AP(MATHLIB)
Fourier	10.59	2.08	22.08
Fourier/Trid	11.25	1.99	22.54
FACR(1)	11.84	1.74	20.56

Table 5-2: Ratios of times (sec) on different architectures for each algorithm using real FFT

percentage of time spent in the normalization from 17% to 8%. (See Table 5-4.) For the 255 x 255 size problem, .16 seconds was saved by the vectorized divide in a normalization that otherwise took .28 seconds, for a 57% reduction in normalization time, and a 10% savings in the entire Poisson problem.

As was discussed in Section 4, the complete Fourier algorithm has the most to gain from an efficient sine transform. We implemented the vectorized sine transform from Section 4 for each

of our algorithms. These results are given in Table 5-5.

		31x31	127x127	511x511
	Preparation	0.5	0.1	0.0
	Sine FFT in X	21.9	20.9	20.7
Fourier	Sine FFT in Y	22.1	21.0	20.7
	Normalization	11.5	16.0	17.1
	Back FFT in Y	22.1	21.0	20.7
	Back FFT in X	21.9	20.9	20.7
	Preparation	0.5	0.1	0.0
	Sine FFT	33.0	28.5	27.5
FFT/Trid	Trid System	30.4	39.8	41.9
(Std Trid)	Back FFT	33.0	28.5	27.5
	Normalization	3.1	3.1	3.0
	Preparation	0.5	0.1	0.0
	Sine FFT	32.6	35.1	37.8
FFT/Trid	Trid System	31.1	25.9	20.3
(New Trid)	Back FFT	32.6	35.1	37.8
	Normalization	3.1	3.9	4.2
	Preparation	0.4	0.1	0.0
	Odd/Even Reduc	8.0	10.6	12.2
	Sine FFT	18.8	20.4	22.6
FACR(1)	Trid System	26.2	22.7	17.5
(New Trid)	Back FFT	18.8	20.4	22.6
	Normalization	5.4	6.1	6.6
	Solve Odd Rows	22.4	19.7	18.5

Table 5-3: % time spent in each algorithm using sine transform, MATHLIB

	Poisson Problem			
	Using Real Transform		Using Sine Transform	
	Fourier	Fourier(Vec)	Fourier	Fourier(Vec)
31x31	.04	.04	.04	.04
63x63	.16	.15	.13	.13
127x127	.62	.58	.46	.43
255x255	2.42	2.27	1.78	1.61
511x511	10.22	9.58	6.76	6.11

Table 5-4: Times times (sec) using Fourier method with Vecdiv and MATHLIB

The vectorised sine transform does result in a noticable improvement in compute time

Poisson Problem

	Fourier	Fourier Vecdiv	Fourier Vecsin	Fourier Vecdiv Vecsin	FACR(1)	FACR(1) Vecsin	F/Trid	F/Trid Vecsin
31x31	.04	.04	.03	.03	.02	.02	.03	.02
63x63	.13	.13	.10	.09	.08	.07	.10	.08
127x127	.46	.43	.37	.33	.24	.21	.29	.25
255x255	1.78	1.61	1.46	1.30	.85	.77	1.07	.95
511x511	6.76	6.11	n/a	n/a	3.10	2.87	3.93	3.52

Table 5-5: Times (sec) using complete Fourier method with optimizations and MATHLIB

although it is more storage intensive. Note however that the FACR(1) method still provides a significant savings in time over the complete Fourier method.

To cut some of the added storage costs, we explored the method of solving m systems instead of all n systems in the FACR(1) algorithm in parallel, for m = 16, 32, and 64. We noted that for loops of size less than 32, the efficiency of the APMATH routines over APFTN optimization level 3 is overshadowed by the overhead of the subroutine calls. We present times for the APFTN-vectorized sine transform in Table 5-6.

Poisson Problem

	block=16	block=32	block=64
31x31	.02	.02	.02
63x63	.08	.07	.07
127x127	.23	.22	.22
255x255	.81	.79	.77
511x511	2.95	2.87	n/a

Table 5-6: Times (sec), blocked-vectorized sine transform, FACR(1) method, MATHLIB

An additional optimization which is important in the case where multiple problems are being solved is the preprocessing of the normalization in the complete Fourier algorithm. Our preprocessing consists of storing the reciprocal of the normalization constants, enabling normalization to be done by a vector multiply. We present times for the preprocessed complete Fourier algorithm in Table 5-7.

We next consider the Fourier/Tridiagonal algorithm. We try to enable the maximum amount of pipelining in the tridiagonal solver by performing the forward substitutions for m

	Poisson Problem preprocessing time	Fourier-vecmul-vecsin excluding preprocessing
31x31	.00	.03
63x63	.01	.09
127x127	.03	.34
255x255	.12	1.22

Table 5-7: Execution times (sec) using Fourier method with preprocessing

systems in parallel followed by the backward substitutions in the solution of the tridiagonal linear systems, cf. Section 3.2. In Table 5-8, we present times for problems of sizes as large as 255 x 255 using the real FFT. Though the method becomes more storage intensive as m grows, we found that our times did not improve unless m was ≥ 31. Note that the improvement in the FACR(1) algorithm is necessarily less since only n/2 tridiagonal linear systems are solved.

Poisson Problem Using Real Transformation	Fourier/Trid	Fourier/Vec-Trid	FACR(1)	FACR(1)/Vec-Trid
31x31	.03	.03	.02	.02
63x63	.10	.09	.08	.08
127x127	.37	.35	.27	.26
255x255	1.39	1.32	1.00	.98

Table 5-8: Times (secs) to solve Poisson's equation, Vectorizing the tridiagonal solves

5.3. Compiler Efficiency

Finally we examine the capability of the architecture of the FPS-164 as demonstrated by use of the MATHLIB routines and the extent to which the existing compiler utilizes it. The times recorded by the APFTN programs vary significantly depending on both the level of optimization of the FPS compiler and the release. Optimization level 3 of Version D04 of the APFTN64 compiler was used in all the tests of this paper. Previous runs had been made using optimization level 2 (due to errors at opt level 3) of the Version C compiler, also at optimization level 2 of the D04 compiler. We present these times for comparison in Table 5-9.

Note that the architecture of the FPS-164 offers a degree of parallelism that is not fully

Fourier/Trid using Sine Transform, APFTN	Opt2,RevC	Opt2,RevD04	Opt3,RevD04
31x31	.10	.05	.04
63x63	.33	.19	.14
127x127	1.16	.67	.50
255x255	4.39	2.44	1.07
511x511	18.29	10.01	5.0

FACR(1) using Sine Transform, APFTN	Opt2,RevC	Opt2,RevD04	Opt3,RevD04
31x31	.04	.04	.03
63x63	.15	.13	.10
127x127	.53	.48	.35
255x255	2.00	1.89	1.23
511x511	8.3	error	4.99

FACR(1) using Sine Transform, MATHLIB	Opt2,RevC	Opt2,RevD04	Opt3,RevD04
31x31	.03	.03	.02
63x63	.10	.09	.08
127x127	.31	.28	.24
255x255	1.10	1.06	.85
511x511	4.03	3.94	3.10

Table 5-9: Execution times (sec) for Version C and Version D APFTN compiler

utilized by any version of the compiler. The sections of the program which make use of the APAL math library improve the running time of the entire Poisson problem by factors of as much as 2, depending on the algorithm. We are expecting research in radical compiler techniques at Yale to provide a FORTRAN compiler that will reduce these factors [6].

We conclude by comparing the best times we were able to calculate for the FPS-164 against the published results obtained for the CDC 7600 by Swarztrauber [11]. We graph the time to solve the Poisson equation for each of our three algorithms, making use of MATHLIB calls and vectorization. These are presented in Figure 5-2.

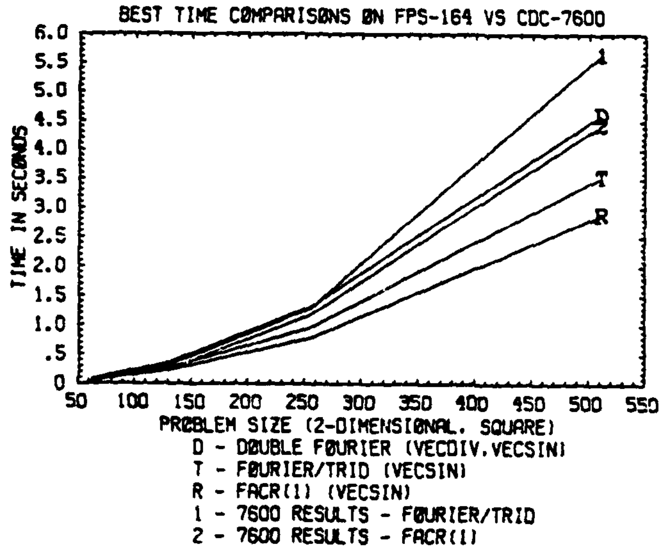


Figure 5-2: Best times for the solution of the Poisson problem on the AP

5.3. Overhead in using the AP

We have found the time required to transfer the data over the Unibus connecting the AP and the VAX to be significant. For the 31x31, 127x127 and 511x511 size problems, the entire overhead including data transfer has been .8, 1.1, and 5.9 seconds. Our real throughput time to transfer the data over the Unibus has been less than one half megabyte per second.

The FPS-164 supports a D64 subsystem which is directly connected to the FPS-164 I/O bus. This has a maximum 1.2 megabyte/second transfer rate, which is comparable to the Unibus transfer rate. One problem of future interest is the out-of-core Poisson problem. See Schultz [10] for an algorithm analysis of this problem. We expect future results which make use of a bulk memory system which we have interfaced to the FPS-164 I/O bus which achieves a 41 megabyte/second transfer rate.

6. Conclusions

Some of the conclusions that should be noted are the following:

- The Malcolm and Palmer technique for the solution of the specialized tridiagonal system of linear equations which results from the Poisson problem is very efficient.
- Efficient implementation of transforms lessen the difference between the complete Fourier, Fourier/Tridiagonal and FACR(1) methods on the FPS-164. However, the most efficient method for the FPS-164 architecture is still the FACR(1) method[5].
- Except for very small problems, in each of the algorithms, utilization of the MATHLIB improves the execution time over FORTRAN code of up to a factor of 2.
- The overhead of data throughput is significant in proportion to the compute time on the FPS-164.
- Optimized code for the FPS-164 has been shown to be more efficient than standard code for the same algorithm on the CDC 7600.

Acknowledgments

We wish to thank Professor Stanley C. Eisenstat for his comments and suggestions.

[5]Our FACR(1) method makes use of the Malcolm and Palmer solution of the tridiagonal system of linear equations.

Appendix A

Upper bound for Convergent Tridiagonal Solver

In Section 3, we outlined the fast tridiagonal solver presented by Malcolm and Palmer [9]. They present upper and lower bounds of convergence of the r_k of equation (28). In this Appendix, we solve the nonlinear difference equation (28) explicitly and subsequently arrive at an equivalent upper bound of convergence to a predefined accuracy.

Let z_1 and z_2 be the solutions of $z = 1/(\lambda - z)$, the equation satisfied by the limit of r_k, such that $|z_1| > |z_2|$. Let $\omega := z_2/z_1$. For $\lambda < -2$, as in the Poisson equation, we have $\lambda < z_1 < z_2 < 0$ and therefore $0 < \omega < 1$.

Clearly

$$z_1 = \frac{\lambda - \sqrt{\lambda^2 - 4}}{2}, \qquad z_2 = \frac{\lambda + \sqrt{\lambda^2 - 4}}{2},$$

$$\lambda = z_1 + z_2 = z_1 (1 + \omega) ,$$

and

$$\omega = z_1^{-2} .$$

It can be shown by induction that

$$r_k = \frac{1 - \omega^k}{z_1(1 - \omega^{k+1})} \tag{32}$$

is the general solution of the recurrence formula (28). This satisfies $r_1 = 1/\lambda$. Given the induction hypothesis, it follows that:

$$r_k = \frac{1}{\lambda - r_{k-1}} = \frac{1 - \omega^k}{\lambda(1 - \omega^k) - z_1^{-1}(1 - \omega^{k-1})}$$

$$= \frac{1 - \omega^k}{z_1(\lambda z_1^{-1} - \lambda z_1^{-1}\omega^k - z_1^{-2} + z_1^{-2}\omega^{k-1})}$$

$$= \frac{1 - \omega^k}{z_1((1 + \omega) - (1 + \omega)\omega^k - \omega + \omega\omega^{k-1})}$$

$$= \frac{1 - \omega^k}{z_1(1 - \omega^{k+1})}$$

Note that $r_k \to 1/z_1 = z_2$ and is monotone for $k \to \infty$.

It is easy to determine the index k_{conv} such that $r_{k_{conv}}$ has converged with maximum relative error ϵ. We determine a simple upper bound, k_0.

$$\left| \frac{r_k - r_\infty}{r_\infty} \right| = \left| \frac{1 - \omega^k}{1 - \omega^{k+1}} - 1 \right|$$

$$= \frac{\omega^k(1 - \omega)}{(1 - \omega^{k+1})}$$

$$\leq \omega^k \leq \epsilon .$$

Therefore

$$k_{conv} \leq k_0 := \left[\frac{\log \epsilon}{\log \omega} \right] . \tag{33}$$

This expression for k_0 is equivalent to the upper bound given by Malcolm and Palmer.

References

[1] Buzbee, B. L., Golub, G. H. and Nielson, C. W. On Direct Methods for Solving Poisson's Equations. *SIAM J. Numer. Anal.* 7:627-656, 1970.

[2] Cooley, J. W., Lewis, P. A. W. and Welch, P. D. The Fast Fourier Transform Algorithm: Programming Considerations in the Calculation of Sine, Cosine and Laplace Transforms. *J. Sound Vib.* 12:350-337, 1970.

[3] Evans, D. J. An Algorithm for the Solution of Certain Tridiagonal Systems of Linear Equations. *Comp. J.* 15:356-359, 1972.

[4] Evans, D. J. and Hatzopoulos, M. The Solution of Certain Banded Systems of Linear Equations Using the Folding Algorithm. *Comp. J.* 19:184-187, 1976.

[5] Fischer, D., Golub, G., Hald, O., Leiva, C. and Widlund, O. On Fourier-Toeplitz Methods for Separable Elliptic Problems. *Math. Comp.* 28:349-368, 1974.

[6] Fisher, J. A. *Very Long Instruction Word Architectures and the ELI-512.* Technical Report #253, Yale University, 1982.

[7] Henrici, P. Fast Fourier Methods in Computational Complex Analysis. *SIAM Review* 21:481-527, 1979.

[8] Hockney, R. W. A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *J. ACM* 12:95-113, 1965.

[9] Malcolm, M. A. and Palmer, J. A Fast Method for Solving a Class of Tridiagonal Linear Systems. *Comm.ACM* 17:14-17, 1974.

[10] Schultz, M. H. *Solving Elliptic Problems on an Array Processor System.* Technical Report YALEU/DSC/RR-272, Yale University, 1983.

[11] Swarztrauber, P. N. The Methods of Cyclic Reduction, Fourier Analysis and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle. *SIAM Review* 19:490-501, 1977.

[12] Temperton, C. Direct Methods for the Solution of the Discrete Poisson Equation: Some Comparisons. *J. Comp. Physics* 31:1-20, 1979.

DAT
FILM